Profile Generation and Guided Optimization for Compiled Queries

Project Proposal Document
April 21, 2021

Abi Kim, Kyle Dotterrer,
Wan Shen Lim
Carnegie Mellon University

1. INTRODUCTION

This document describes our milestone report for our final project in CMU's 15-745: *Optimizing Compilers* course. The remainder of this document is structured as follows. First, in Section 1, we describe the major changes that we have introduced since our original proposal. Next, in Section 2, we present our current progress. Section 3 discusses our progress relative to our original milestone goal. In Section 5 we list some of the surprises we have encountered thuis far. Section 6 describes our revised schedule for the remainder of the project. We conclude in Section 7 with our required resources.

2. MAJOR CHANGES

One major change that has occurred since our initial proposal is the publication of a paper by the TUM group that has implications for our project [2]. This paper demonstrates that it is possible to gather profiling information across multiple abstraction levels, specifically, across the query compilation boundary. This was a major open question in the feasibility of our proposal that is now (in theory) resolved. Furthermore, the paper describes some high-level implementation details of the mechanisms used to gather and process this profiling information, giving us answers to several design decisions that we had not yet resolved. With the publication of this paper, we may proceed under the assumption that we can record profile information for JIT-compiled queries and subsequently attach the recorded metrics (e.g. execution time, instruction count, memory hierarchy behavior, etc.) to any layer of the query compilation pipeline (e.g. bytecode instruction, DSL operation, or SQL operator).

Profile generation at arbitrary abstraction levels is the primary focus of the TUM group's recent publication. In this model (and in the example use cases presented in the paper), the profile information is intended to grant developers further insights into the behavior of the code in their dataflow system, allowing them to manually inspect the source of performance limitations and optimize accordingly. Therefore, the portion of our project that addresses automatic selection and application of optimization passes based on the profiling information remains a novel and interesting direction for research. For this reason, our updated goals for the project are to:

- Implement a profiling subsystem similar to the one described in the TUM group paper
- Utilize the profile data generated by this subsystem to inform automatic tuning and selection of optimization passes
- Ideally, characterize the effect of typical optimization passes for different database-centric workloads

This set of goals replaces one of the original goals we formulated in our project proposal with the last new goal. Specifically, we no longer endeavor to implement a new custom LLVM pass which would operate on the collected profile data and apply it to the lowest-layer of the NoisePage [1] execution engine. There are a number of reasons we have revised our goals to exclude this requirement:

- Development Time: While the description of profile generation by the TUM paper certainly helps us by answering some design questions, it simultaneously introduces significant implementation complexity. We estimate that implementing the approach described in the paper will prove to be significantly more difficult than the approach we initially intended to pursue. We believe however that the additional effort involved will be worth the cost.
- Reuse: Profile-guided optimization already exists to a certain degree in LLVM, and it seems more appropriate to try to hook into the existing LLVM infrastructure than to come up with a domain-specific PGO pass.
- Value: It is possible for us to work more quickly in the time remaining by aiming to apply existing optimization passes at a higher level of abstraction, such as at the level of "what LLVM optimization passes make sense for an OLTP workload vs an OLAP workload". Prior work in e.g., ML has shown that the order in which passes are applied matters, and therefore we aim to focus on the same in a database context.

3. CURRENT PROGRESS

This section describes the progress that we have made so far. Please see the appendix for a simple example.

3.1 Profile Generation

In our first attempt at profile generation, we were able to implement a system in which we can gather pipeline-level execution times and function-level optimization times.

Since reading the TUM paper, we have since revised our approach to profile generation. Specifically, we now have a complete, concrete plan for gathering execution profiles for JIT-compiled queries in NoisePage that proceeds approximately as follows:

- Track the query through stages of query compilation. When a query is submitted to the system, (and we desire profile generation for the query, naturally this cannot be the default mode for performance reasons), we track the query through each stage of query compilation and generate a tagging dictionary [0] at each stage. This tagging dictionary effectively maps from the lower-level intermediate representation for the query to the higher-level intermediate representation construct that generated it.

- Wrap the execution of the JIT-compiled query in the necessary profiling infrastructure to track desired hardware and software metrics. Currently, much of the infrastructure necessary to collect a fine-grained execution profile for query execution is tied to the Linux perf API.

- Combine the profile data with the tagging dictionaries for the query to map from the native code back up to the desired abstraction level. In order to implement the **EXPLAIN ANALYZE** command, we map all the way back up to the operator tree for the query, but there is no reason that we cannot also present profile information at lower levels of the query compilation abstraction hierarchy.

3.2 Feedback Loop

Most of our work on the other major component of our project has consisted of literature review on possible optimizations that we might apply during the automated tuning process, and additional constraints that the existing codebase(s) may impose on us. We have not yet begun integrating the code that implements the automated tuning framework into the NoisePage codebase.

4. MEETING OUR MILESTONE

We did not meet the milestone identified in our original project proposal. The primary reason for this is that we re-prioritized the overall goals of our project on the basis of new information:

- Further consideration of the merits of implementing profile-guided optimization passes in the context of a data-centric DBMS execution engine (see *Major Changes*)

- The publication of the TUM paper describing profile generation across abstraction levels in dataflow systems

In other words, we did not meet our milestone, but this is largely a result of the fact that our project goals have shifted significantly. We recognize that this still constitutes a shortcoming of our initial planning for the project, but we stand by the architectural changes we have made and are confident in both our direction and rate of progress.

5. SURPRISES

One implementation-related surprise we have encountered pertains to the LLVM API. It turns out that LLVM offers both legacy and new-style pass managers, where the new-style pass managers appear to have slightly incomplete support for certain legacy patterns, and where our (fixed) environment appears to be missing certain dependencies required to use the new-style pass managers. Initially it appeared that it would be beneficial to switch to the new-style pass manager for the purpose of instrumenting *CFG simplification* time at a finer granularity, but it appears that we will have to pursue an alternative approach.

6. REVISED SCHEDULE

The revised schedule for each of our group members is provided below.

- **Abi**: Will continue working on the feedback loop jointly with Wan (formerly with Kyle).

- **Wan**: Was initially working on instrumentation for the execution engine. In light of the TUM paper and shifting interests, will now work on the automated optimization selection and tuning framework (feedback loop). For an example of feedback loop ideas and direction, please see appendix.

- **Kyle**: Was initially working on the automated optimization selection and tuning framework. In light of the TUM paper and shifting interests, will now work on implementing the profiling subsystem described in the TUM paper.

7. REQUIRED RESOURCES

We do not require any additional resources to complete our project.

APPENDIX

A. OPTIMIZATION FOR COMPILED QUERIES

Currently, Prof. Mowry's Ph.D. student Prashanth has empirically identified the below combination of passes as the best combination of passes for query optimization for his thesis work. This combination of passes is uniformly applied to all queries that we compile in the NoisePage DBMS.

```
// Run the following at O3, no size optimization, inline at
     hot call sites.
//   Instruction recombine
//   Reassociation
//   Global value numbering
//   Control-flow-graph simplification
//   Aggressive dead-code elimination
//   Control-flow-graph simplification
```

You may however imagine using knowledge such as workload type (OLTP vs OLAP, i.e., short running vs long lived, point updates vs range scans, etc.), to pick a different set of function passes for different queries, etc. After consideration of our options and the constraints of the existing code, this is an example of the domain-specific knowledge that we believe can be incorporated into our feedback loop, in addition to the typical profile statistics of both compilation and execution time.

A simple baseline for understanding what the passes are doing is to remove all passes, checking to see the quality of the subsequent IR. This was previously done by hand. We have invested time in the project so far in making it simple to quickly obtain targeted information about a function. This is demonstrated here with a small excerpt, serving also as a demo of some information that we are currently gathering (note that this only shows information gathered at compile time, information such as execution time and cache misses which need to be gathered at run time is currently harder to instrument).

A.1 Input DSL

The DSL for performing an aggregation over a table that is being compiled is shown in the listing below.

```
fun pipeline_1(execCtx: *ExecutionContext, state: *State)
     -> nil {
   var ht = &state.table
   var tvi: TableVectorIterator
   var table_oid = @testCatalogLookup(execCtx, "test_1",
        "")
   var col_oids: [2]uint32
   col_oids[0] = @testCatalogLookup(execCtx, "test_1",
        "colA")
```

```
    col_oids[1] = @testCatalogLookup(execCtx, "test_1",
         "colB")
    for (@tableIterInit(&tvi, execCtx, table_oid,
         col_oids); @tableIterAdvance(&tvi); ) {
        var vec = @tableIterGetVPI(&tvi)
        for (; @vpiHasNext(vec); @vpiAdvance(vec)) {
            var hash_val = @hash(@vpiGetInt(vec, 1))
            var agg = @ptrCast(*Agg, @aggHTLookup(ht,
                 hash_val, keyCheck, vec))
            if (agg == nil) {
                agg = @ptrCast(*Agg, @aggHTInsert(ht,
                     hash_val))
                constructAgg(agg, vec)
            } else {
                updateAgg(agg, vec)
            }
        }
    }
    @tableIterClose(&tvi)
}
```

A.2 Optimizations Applied

With all of Prashanth's optimizations applied, the third basic block becomes

```
[2021-04-21 21:21:51.175] [execution_logger] [info] LLVM
     dump for pipeline_1 (90 instructions, 551831 ns
     optimize):

define void
     @pipeline_1(%"class.noisepage::execution::exec::ExecutionContext"*,
     %0*) personality i32 (...)* @__gxx_personality_v0 {

BB3:                                             ; preds =
     %BB7, %BB2
  %10 = phi i16 [ %67, %BB7 ], [ %.pre, %BB2 ]
  %11 = getelementptr inbounds
       %"class.noisepage::execution::sql::TableVectorIterator",
       %"class.noisepage::execution::sql::TableVectorIterator"*
       %2, i64 0, i32 7, i32 3
  %12 = load i16, i16* %11, align 8, !tbaa !9
  %13 = icmp ult i16 %10, %12
  br i1 %13, label %BB4, label %BB1
```

A.3 Without Optimization

With no optimizations applied, the third basic block appears as follows

```
[2021-04-21 21:20:57.958] [execution_logger] [info] LLVM
     dump for pipeline_1 (105 instructions, 118338 ns
     optimize):

define void
     @pipeline_1(%"class.noisepage::execution::exec::ExecutionContext"*,
     %0*) personality i32 (...)* @__gxx_personality_v0 {

...

BB3:                                             ; preds =
     %BB7, %BB2
  %12 = bitcast
       %"class.noisepage::execution::sql::TableVectorIterator"*
       %2 to i8*
  %sunkaddr = getelementptr i8, i8* %12, i64 4290
  %13 = bitcast i8* %sunkaddr to i16*
  %14 = load i16, i16* %13, align 2, !tbaa !2
  %15 = zext i16 %14 to i64
  %16 = bitcast
       %"class.noisepage::execution::sql::TableVectorIterator"*
       %2 to i8*
  %sunkaddr18 = getelementptr i8, i8* %16, i64 4288
  %17 = bitcast i8* %sunkaddr18 to i16*
  %18 = load i16, i16* %17, align 8, !tbaa !9
  %19 = icmp ult i16 %14, %18
  %20 = zext i1 %19 to i8
  br i1 %19, label %BB4, label %BB1
```

A.4 Discussion

As shown, we are able to obtain simple statistics like instruction count, time taken to apply optimization passes, etc.

B. REFERENCES

[1] NoisePage. https://noise.page.

[2] A. Beischl, T. Kersten, M. Bandle, J. Giceva, and T. Neumann. Profiling dataflow systems on multiple abstraction levels, 2021.